**Strings and basics of text processing**

**Strings**

1. *Как создать объект класса String, какие конструкторы класса String вы знаете? Что такое строковый литерал? Объясните, что значит “упрощенное создание объекта String”?*

В Java строка создается следующими способами:

1. String strA = “java”;

2. String strB = new String(“java”);

Конструкторы:

* **String()** — создается объект с пустой строкой;
* **String(String str)** — конструктор копирования: из одного объекта создается его точная копия, поэтому данный конструктор используется редко;
* **String(StringBuffer str)** — преобразованная копия объекта класса StringBuffer;
* **String(StringBuilder str)** — преобразованная копия объекта класса StringBuilder;
* **String(byte[] byteArray)** — объект создается из массива байтов byteArray;
* **String(char[] charArray)** — объект создается из массива charArray символов Unicode;
* **String(byte[] byteArray, int offset, int count)** — объект создается из части массива байтов byteArray, начинающейся с индекса offset и содержащей count байтов;
* **String(char[] charArray, int offset, int count)** — то же, но массив состоит из символов Unicode;
* **String(int[] intArray, int offset, int count)** — то же, но массив состоит из символов Unicode, записанных в массив целого типа, что позволяет использовать символы Unicode, занимающие больше двух байтов;
* **String(byte[] byteArray, String encoding)** — символы, записанные в массиве байтов, задаются в Unicode-строке с учетом кодировки encoding;
* **String(byte[] byteArray, int offset, int count, String encoding)** — то же самое, но только для части массива;
* **String(byte[] byteArray, Charset charset)** — символы, записанные в массиве байтов, задаются в Unicode-строке с учетом кодировки, заданной объектом charset;
* **String(byte[] byteArray, int offset, int count, Charset charset)** — то же самое, но только для части массива.

Строковые литералы — это набор символов, заключенных в двойные кавычки. Данный тип используется так же часто, как и числовые литералы. В строке также могут находится служебные символы, которые необходимо экранировать (так называемые escape-последовательности).

Упрощенная создание объекта String – это создание без использования оператора new: String strA = “java”.

1. *Можно ли изменить состояние объекта типа String? Что происходит при попытке изменения состояния объекта типа String? Можно ли наследоваться от класса String? Как вы думаете, почему строковые объекты immutable?*

Объекты типа String неизменяемы. Если String-объект создан, то его содержимое не может быть изменено. Хотя данное утверждение похоже на серьезное ограничение, однако это не так по двум причинам:

- если нужно изменить строку, то всегда можно создать её новую модифицированную копию;

- Java определяет класс просмотра (peer class) для String, называемый StringBuffer, который позволяет строкам стать изменяемыми, так что все нормальные манипуляции со строками все еще доступны в Java.

Под неизменяемостью, или иммутабельностью (immutability) подразумевается невозможность изменения содержимого объекта после его создания.

Каждая строка это уникальный объект, если изменяется строка значит переменной подставляется ссылка на совсем другой объект типа String.

String – это final класс, нельзя от него наследоваться.

У неизменности строк есть ряд неоспоримых преимуществ:

* *Строковый пул (String pool)* возможен только благодаря тому, что строки в ***Java*** неизменяемы. Виртуальная машина имеет возможность сохранить много места в памяти (*heap space*) т.к. разные строковые переменные указывают на одну переменную в пуле. При изменяемости строк было бы невозможно реализовать интернирование, поскольку если какая-либо переменная изменит значение, это отразится также и на остальных переменных, ссылающихся на эту строку.
* Изменяемость строк несло бы в себе потенциальную угрозу безопасности приложения. Поскольку в ***Java*** строки используются для передачи параметров для авторизации, открытия файлов и т.д. — неизменяемость позволяет избежать проблем с доступом.
* Так как строка неизменяемая то, она безопасна для много поточности и один экземпляр строки может быть совместно использован различными потоками. Это позволяет избежать синхронизации для потокобезопасности. Таким образом, строки в ***Java*** полностью потокобезопасны.
* Поскольку строка неизменная, её *hashcode* кэшируется в момент создания и нет никакой необходимости рассчитывать его снова. Это делает строку отличным кандидатом для ключа в ***Map*** и его обработка будет быстрее, чем других ключей ***HashMap***. Поэтому строка наиболее часто используется в качестве ключа ***HashMap***.

 Если резюмировать вышесказанное, то получаем, что основные причины неизменяемости ***String*** в ***Java*** это безопасность и наличие пула строк (*String pool*).

* можно передавать строку между потоками не опасаясь, что она будет изменена
* отсутствуют проблемы с синхронизацией потоков
* отсутствие проблем с  утечкой памяти
* отсутствие проблем с доступом и безопасностью при использовании строк для передачи параметров авторизации, открытия файлов и т.д.
* кэширование *hashcode*
* Экономия памяти при использовании пула строк для хранения повторяющихся строк.

1. *Объясните, что такое кодировка? Какие кодировки вы знаете? Как создать строки в различной кодировке?*

Кодировка - это таблица, где каждой букве/символу соответствует число. Наиболее распространённые кодировки – это ASCII и UTF, также существуют менее распространенные кодировки - Windows-1252, KOI8-R, CP 866, ISO 8859-5.

Для создания строк в кодировке используется конструктор:

* **String(byte[] byteArray, Charset charset)** — символы, записанные в массиве байтов, задаются в Unicode-строке с учетом кодировки, заданной объектом charset;
* **String(byte[] byteArray, String encoding)** — символы, записанные в массиве байтов, задаются в Unicode-строке с учетом кодировки encoding;

Тип кодировки берется из таблиц: UTF8, ASCII, ISO8859\_1, Cp866 и др.

1. *Что такое пул литералов? Как строки заносятся в пул литералов? Как занести строку в пул литералов и как получить ссылку на строку, хранящуюся в пуле литералов? Где хранится (в каком типе памяти) пул литералов в Java 1.6 и Java 1.7?*

Пул строк (**String Pool**) — это множество строк в кучи ([Java Heap Memory](https://javadevblog.com/chto-takoe-heap-i-stack-pamyat-v-java.html" \t "_blank)). Сам строковый пул возможен только потому, что [строки в Java неизменные](https://javadevblog.com/pochemu-stroki-v-java-neizmennye.html). Также пул строк позволяет сохранить память в Java Runtime, хотя это и требует больше времени на создание самой строки.

Когда мы используем двойные кавычки, чтобы создать новую строку, то первым делом идет поиск строки с таким же значением в пуле строк. Если java такую строку нашла, то возвращает ссылку, в противном случае создается новая строка в пуле, а затем возвращается ссылка.

Однако использование оператора new заставляет класс String создать новый объект String. После этого можем использовать метод intern(), чтобы поместить этот объект в пул строк или обратиться к другому объекту из пула строк, который имеет такое же значение.

До Java 1.7 JVM помещал пул Java String в пространство PermGen , которое имеет фиксированный размер - его нельзя развернуть во время выполнения и не подходит для сборки мусора .

Риск интернирования Strings в PermGen (вместо Heap) заключается в том, что мы можем получить ошибку OutOfMemory от JVM, если мы интернируем слишком много Strings .

Начиная с Java 1.7, пул Java String хранится в пространстве Heap, которое собирает мусор с помощью JVM. Преимущество этого подхода заключается в уменьшенном риске ошибки OutOfMemory, поскольку несвязанные Strings будут удалены из пула , тем самым освобождая память.

1. *В чем отличие объектов классов StringBuilder и StringBuffer от объектов класса String? Какой из этих классов потокобезопасный? Как необходимо сравнивать на равенство объекты классов StringBuilder и StringBuffer и почему?*

Объекты String являются неизменяемыми, поэтому все операции, которые изменяют строки, фактически приводят к созданию новой строки, что сказывается на производительности приложения. Для решения этой проблемы, чтобы работа со строками проходила с меньшими издержками в Java были добавлены классы StringBuffer и StringBuilder. По сути они напоминает расширяемую строку, которую можно изменять без ущерба для производительности.

Эти классы похожи, практически двойники, они имеют одинаковые конструкторы, одни и те же методы, которые одинаково используются. Единственное их различие состоит в том, что класс StringBuffer синхронизированный и потокобезопасный. То есть класс StringBuffer удобнее использовать в многопоточных приложениях, где объект данного класса может меняться в различных потоках. Если же речь о многопоточных приложениях не идет, то лучше использовать класс StringBuilder, который не потокобезопасный, но при этом работает быстрее, чем StringBuffer в однопоточных приложениях.

Внутри класса String метод equals переопределён, поэтому можно сравнивать строки по ЗНАЧЕНИЮ с помощью equals, а в классе StringBuilder нет. Метод equals StringBuffer не переопределяется из Object, поэтому это просто ссылочное равенство, то есть то же, что и при использовании ==. Причина этого в том, что StringBuffer является изменяемым, а переопределение equals в основном полезно для классов, подобных значению, которые вы можете использовать в качестве ключей (хотя в списках также есть переопределенные equals и StringBuffer вид списка, так что это немного непоследовательно). Чтобы сравнить через equals объект StringBuffer , необходимо его привести к объекту типа String помощью toString.

1. *Что такое Unicode?*

Юникод (чаще всего) или Уникод ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Unicode) — стандарт [кодирования символов](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B1%D0%BE%D1%80_%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D0%BE%D0%B2), включающий в себя знаки почти всех письменных [языков](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA) мира. В настоящее время стандарт является преобладающим в [Интернете](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82).

Стандарт предложен в [1991 году](https://ru.wikipedia.org/wiki/1991_%D0%B3%D0%BE%D0%B4) некоммерческой организацией «Консорциум Юникода» ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Unicode Consortium, Unicode Inc.). Применение этого стандарта позволяет закодировать очень большое число символов из разных систем письменности: в документах, закодированных по стандарту Юникод, могут соседствовать китайские [иероглифы](https://ru.wikipedia.org/wiki/%D0%98%D0%B5%D1%80%D0%BE%D0%B3%D0%BB%D0%B8%D1%84), математические символы, буквы [греческого алфавита](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B5%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BB%D1%84%D0%B0%D0%B2%D0%B8%D1%82), [латиницы](https://ru.wikipedia.org/wiki/%D0%9B%D0%B0%D1%82%D0%B8%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BB%D1%84%D0%B0%D0%B2%D0%B8%D1%82) и [кириллицы](https://ru.wikipedia.org/wiki/%D0%9A%D0%B8%D1%80%D0%B8%D0%BB%D0%BB%D0%B8%D1%86%D0%B0), символы музыкальной нотной нотации, при этом становится ненужным переключение [кодовых страниц](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%86%D0%B0).

Стандарт состоит из двух основных частей: универсального набора символов ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Universal character set, UCS) и семейства кодировок ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Unicode transformation format, UTF). Универсальный набор символов перечисляет допустимые по стандарту Юникод символы и присваивает каждому символу код в виде неотрицательного целого числа, записываемого обычно в шестнадцатеричной форме с префиксом U+, например, U+040F. Семейство кодировок определяет способы преобразования кодов символов для передачи в потоке или в файле.

Коды в стандарте Юникод разделены на несколько областей. Область с кодами от U+0000 до U+007F содержит символы набора [ASCII](https://ru.wikipedia.org/wiki/ASCII), и коды этих символов совпадают с их кодами в ASCII. Далее расположены области символов других систем письменности, знаки пунктуации и технические символы. Часть кодов зарезервирована для использования в будущем. Под символы кириллицы выделены области знаков с кодами от U+0400 до U+052F, от U+2DE0 до U+2DFF, от U+A640 до U+A69F

1. *Какие методы класса String используются для работы с кодовыми точками? Как вы думаете, когда следует их использовать?*

В языке Java строки реализованы как последовательности значений типа char. Тип char позволяет задавать кодовые единицы, представляющие кодовые точки Unicode в кодировке UTF-16. Наиболее часто используемые символы Unicode представляются одной кодовой единицей. Дополнительные символы задаются парами кодовых единиц.

Метод length() возвращает количество кодовых единиц для данной строки в кодировке UTF-16.

Чтобы определить реальную длину, представляющую собой число кодовых точек, надо использовать следующий вызов:

int cpCount = greeting.codePointCount(0, greeting.length());

**Метод charAt(n)** возвращает кодовую единицу в позиции n, где n находится в интервале от 0 до length() - 1.

*char first = greeting.charAt(0); // первый символ - 'H'*

*char last = greeting.charAt(4); // последний символ - 'o'*

Получения i-й кодовой точки:

*int index = greeting.offsetByCodePoints(0, i);*

*int cp = greeting.codePointAt(index);*

Java подсчитывает кодовые единицы в строках специфическим образом: первая кодовая единица в строке расположена в позиции 0.

Для представления символа Z используются две кодовые единицы UTF-16. Приведенный ниже вызов метода даст не код пробела, а второй код символа Z. Чтобы избежать возникновения данной проблемы, не следует применять тип char, так как он представляет символы на слишком низком уровне.

Если необходимо просмотреть строку посимвольно, то есть получить по очереди каждую кодовую точку, надо использовать фрагмент кода, подобный показанному ниже.

*int cp = sentence.codePointAt(i);*

*if(Character.isSupplementaryCodePoint(cp)) i+=2;*

*eslt i++;*

**Метод codePointAt()** определяет, является ли кодовая единица первой или второй частью дополнительного символа, и всегда возвращает корректный результат. Поэтому можно организовать просмотр строки и в обратном направлении.

**Regular Expressions**

1. *Расскажите, что представляет собой регулярное выражение? Что такое метасимволы регулярного выражения? Какие вы знаете классы символов регулярных выражений? Что такое квантификаторы? Какие логические операторы регулярных выражений вы знаете? Что значит “якорь” для регулярного выражения?*

Регулярное выражение (RegEx) – это шаблон для поиска строки в тексте. В Java исходным представлением этого шаблона всегда является строка, то есть объект класса String. Однако не любая строка может быть скомпилирована в регулярное выражение, а только та, которая соответствует правилам написания регулярного выражения – синтаксису, определенному в спецификации языка. Для написания регулярного выражения используются буквенные и цифровые символы, а также метасимволы – символы, имеющие специальное значение в синтаксисе регулярных выражений.

Чтобы создать RegEx в Java, нужно сделать два простых шага:

- написать его в виде строки с учётом синтаксиса регулярных выражений;

- скомпилировать эту строку в регулярное выражение;

Работа с регулярными выражениями в любой Java-программе начинается с создания объекта класса Pattern. Для этого необходимо вызвать один из двух имеющихся в классе статических методов compile. Первый метод принимает один аргумент – строковый литерал регулярного выражения, а второй – плюс еще параметр, включающий режим сравнения шаблона с текстом:

**public** **static** Pattern compile (String literal)

**public** **static** Pattern compile (String literal, **int** flags)

[Список](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#CASE_INSENSITIVE) возможных значений параметра flags определен в классе Pattern и доступен нам как статические переменные класса.

Pattern pattern = Pattern.compile("java", Pattern.CASE\_INSENSITIVE);

//поиск совпадений с шаблоном будет производиться без учета регистра символов.

По сути, класс Pattern — это конструктор регулярных выражений. Под «капотом» метод compile вызывает закрытый конструктор класса Pattern для создания скомпилированного представления. Такой способ создания экземпляра шаблона реализован с целью создания его в виде неизменяемого объекта. При создании производится синтаксическая проверка регулярного выражения. При наличии ошибок в строке – генерируется исключение PatternSyntaxException.

***Синтаксис регулярных выражений***

Синтаксис регулярных выражений основан на использовании символов < ( [ { \ ^ - = $ ! | ] } ) ? \* + . > , которые можно комбинировать с буквенными символами. В зависимости от роли их можно разделить на несколько групп:

1. Метасимволы для поиска совпадений границ строк или текста

|  |  |
| --- | --- |
| **Метасимвол** | **Назначение** |
| ^ | начало строки |
| $ | конец строки |
| \b | граница слова |
| \B | не граница слова |
| \A | начало ввода |
| \G | конец предыдущего совпадения |
| \Z | конец ввода |
| \z | конец ввода |

2. Метасимволы для поиска символьных классов

|  |  |
| --- | --- |
| **Метасимвол** | **Назначение** |
| \d | цифровой символ [0-9] |
| \D | нецифровой символ [^\d] |
| \s | символ пробела [\t \n \f \r \p {Z}] |
| \S | непробельный символ [^\s] |
| \w | буквенно-цифровой символ или знак подчёркивания [a-zA-Z0-9\_] |
| \W | любой символ, кроме буквенного, цифрового или знака подчёркивания [^\w] |
| . | любой символ, кроме новой строки |

3. Метасимволы для поиска символов редактирования текста

|  |  |
| --- | --- |
| **Метасимвол** | **Назначение** |
| \t | символ табуляции |
| \n | символ новой строки |
| \r | символ возврата каретки |
| \f | переход на новую страницу |
| \u 0085 | символ следующей строки |
| \u 2028 | символ разделения строк |
| \u 2029 | символ разделения абзацев |

4. Метасимволы для группировки символов

|  |  |
| --- | --- |
| **Метасимвол** | **Назначение** |
| [абв] | любой из перечисленных (а,б, или в) |
| [^абв] | любой, кроме перечисленных (не а,б, в) |
| [a-zA-Z] | слияние диапазонов (латинские символы от a до z без учета регистра ) |
| [a-d[m-p]] | объединение символов (от a до d и от m до p) |
| [a-z&&[def]] | пересечение символов (символы d,e,f) |
| [a-z&&[^bc]] | вычитание символов (символы a, d-z) |

5. Метасимволы для обозначения количества символов – квантификаторы. Квантификатор всегда следует после символа или группы символов.

|  |  |
| --- | --- |
| **Метасимвол** | **Назначение** |
| ? | один или отсутствует |
| \* | ноль или более раз |
| + | один или более раз |
| {n} | n раз |
| {n,} | n раз и более |
| {n,m} | не менее n раз и не более m раз |

*Точка*

. - это простейший пример метасимвола. Метасимвол . соответствует любому единичному символу. Например, регулярное выражение .ar означает: любой символ, за которым следует буква a, за которой следует буква r.

«.ar» => The car parked in the garage.

*Интервал символов*

Интервал или набор символов также называют символьным классом. Для его обозначения используются квадратные скобки. Чтобы указать диапазон символов внутри класса, необходимо поставить знак тире. Порядок ряда символов в наборе неважен. Так, например, регулярное выражение [Tt]he означает: T или t, за которым следует буква h, за которой следует буква e.

«[Tt]he » => The car parked in the garage.

Стоит отметить, что точка, помещенная в квадратные скобки, означает именно точку, а ничто другое. Таким образом регулярное выражение ar[.] означает строчный символ a, за которым следует буква r, за которой следует точка..

«ar [.]» => A garage is a good place to park a car.

*Отрицание набора символов*

Обычно символ ^ представляет начало строки, но когда он внутри квадратных скобок, все символы, которые находятся после него, исключаются из шаблона. Например, выражение [^c]ar поможет отыскать все символы кроме c, за которыми следуют а и r.

"[^c]ar" => The car parked in the garage.

*Повторения*

Следующие мета-символы + ,\* или ? используются для того, чтобы обозначить допустимое количество повторения подшаблона. Их роль зависит от конкретного случая.

*Звездочка \**

Этот символ поможет найти одно или более копий какого-либо символа. Регулярное выражение a\* означает 0 или более повторений символа a. Но если этот символ появится после набора или класса символов, тогда будут найдены повторения всего сета. Например, выражение [a-z]\* означает любое количество этих символов в строке.

"[a-z]\*" => The car parked in the garage #21.

Также символ может быть использован вместе с метасимволом . для подбора строки из любых символов .\*.

Еще звездочку можно использовать со знаком пробела \s, чтобы подобрать строку из пробелов. Например, выражение \s\*cat\s будет означать 0 или более пробелов, за которыми следует символ с, за ним а и t, а за ними снова 0 либо больше пробелов.

"\s\*cat\s\*" => The fat cat sat on the concatenation.

*Плюс +*

+ соответствует одному или нескольким повторениям предыдущего символа. Например, регулярное выражение c.+t означает: строчная буква c, за которой следует хотя бы один символ, за которым следует строчный символ t. Необходимо уточнить, что буква t должна быть последней t в предложении.

"c.+t" => The fat cat sat on the mat.

*Вопросительный знак ?*

В regexp метасимвол ? делает предшествующий символ необязательным. Этот символ соответствует полному отсутствию или же одному экземпляру предыдущего символа. Например, регулярное выражение [T]?he означает: необязательно заглавную букву T, за которой следует строчный символ h, за которым следует строчный символ e.

"[T]he" => The car is parked in the garage.

"[T]?he" => The car is parked in the garage.

*Скобки*

Скобки в regexp, которые также называются квантификаторами, используются для указания допустимого количества повторов символа или группы символов. Например, регулярное выражение [0-9]{2,3} означает, что допустимое количество цифр должно быть не менее двух цифр, но не более 3 (символы в диапазоне от 0 до 9).

"[0-9]{2,3}" => The number was 9.9997 but we rounded it off to 10.0.

Мы можем убрать второе число. Например, выражение [0-9]{2,} означает 2 или более цифр. Если мы также уберем запятую, то тогда выражение [0-9]{3} будет находить только лишь 3 цифры, ни меньше и ни больше.

"[0-9]{2,}" => The number was 9.9997 but we rounded it off to 10.0.

"[0-9]{3}" => The number was 9.9997 but rounded it off to 10.0.

*Символьная группа*

Группа символов - это группа подшаблонов, которая записывается внутри скобок (...). Как было упомянуто раньше, если в регулярном выражении поместить квантификатор после символа, он повторит предыдущий символ. Но если мы поставим квантификатор после группы символов, он просто повторит всю группу. Например, регулярное выражение (ab)\* соответствует нулю или более повторениям символа «ab». Мы также можем использовать | - метасимвол чередования внутри группы символов. Например, регулярное выражение (c|g|p)ar означает: символ нижнего регистра c, g или p, за которым следует символ a, за которым следует символ r.

"(c|g|p)ar" => The car is parked in the garage.

*Перечисление*

В regexp вертикальная полоса | используется для определения перечисления. Перечисление - это что-то вроде условия между несколькими выражениями. Можно подумать, что набор символов и перечисление работают одинаково, но это совсем не так, между ними существует огромная разница. Перечисление работает на уровне выражений, а набор символов на уровне знаков. Например, регулярное выражение (T|t)he|car означает: T или t, сопровождаемая строчным символом h, сопровождаемый строчным символом e или строчным символом c, а затем a и r.

"(T|t)he|car" => The car is parked in the garage.

*Исключение специального символа*

Обратная косая черта \ используется в regexp, чтобы избежать символа, который следует за ней. Это позволяет нам указывать символ в качестве символа соответствия, включая зарезервированные { } [ ] / \ + \* . $ ^ | ?. Чтобы использовать специальный символ в качестве подходящего, перед ним нужно поставить \.

Например, регулярное выражение . используется для нахождения любого единичного символа. Регулярное выражение (f|c|m)at\.? означает строчную букву f, c или m, а затем a, за ней t с последующим дополнительным символом ..

"(f|c|m)at\.?" => The fat cat sat on the mat.

*Анкеры - Привязки*

В regexp мы используем привязки, чтобы проверить, является ли соответствующий символ первым или последним символом входной строки. Привязка бывает двух типов: первый - это ^, который проверяет является ли соответствующий символ первым введенным, а второй - знак доллара, который проверяет, является ли соответствующий символ последним символом введенной строки.

*Caret*

Символ ^ используется в regexp, чтобы проверить, является ли соответствующий символ первым символом в введенной строке. Если мы применяем следующее регулярное выражение ^a (проверяем является ли a первым символом) для введенной строки abc, то оно будет равно a. Но если мы применим регулярное выражение ^b к той же строке, то оно ничего не вернет, потому что во входной строке abc символ «b» не является первым. Давайте посмотрим на другое регулярное выражение ^(T|t)he, которое означает: T или t - это символ начала входной строки, за которым следует строчный символ h, а затем e.

"(T|t)he" => The car is parked in the garage.

"^(T|t)he" => The car is parked in the garage.

*Доллар*

Знак доллара используется для проверки, является ли символ в выражении последним в введенной строке. Например (at\.)$ означает строчную а, за которой следует t, за которой следует a ., которые должны заканчивать строку.

"(at\.)" => The fat cat. sat. on the mat.

"(at\.)$" => The fat cat. sat. on the mat.

У символов каретки ^ и доллара $ есть специальные значения в регулярных выражениях. Они называются «якоря» (anchors). Каретка ^ означает совпадение с началом текста, а доллар $ – с концом. Якоря ^ и $ – это проверки. У них нулевая ширина.

Другими словами, они не добавляют к результату поиска символы, а только заставляют движок регулярных выражений проверять условие (начало/конец текста).

***Lookaround Позиционная проверка***

Lookbehind и lookahead (также называемые lookaround) - это определенные типы non-capturing групп (Они используются для поиска, но сами в него не входят). Lookaheads используются, когда у нас есть условие, что этому шаблону предшествует или следует другой шаблон. Например, мы хотим получить все числа, которым предшествует символ $ из входной строки $4.44 and $10.88. Мы будем использовать регулярное выражение (?<=\$)[0-9\.]\*, которое означает: получить все числа, содержащие . и которым предшествует символ $. Ниже приведены lookarounds, что используются в регулярных выражениях:

| Символ | Описание |
| --- | --- |
| ?= | Положительный Lookahead |
| ?! | Отрицательный Lookahead |
| ?<= | Положительный Lookbehind |
| ?<! | Отрицательный Lookbehind |

*Положительный Lookahead*

Положительный lookahead означает, что эта часть выражения должна следовать за впереди идущим выражением. Возвращаемое значение содержит текст, который совпадает с первой частью выражения. Чтобы определить позитивный lookahead, используют скобки. Внутри них размещают знак вопроса и знак равенства: (?=...). Само же выражение пишется после =. Например, выражение (T|t)he(?=\sfat) - это T в верхнем или нижнем регистре, за которым следует h и e. В скобках мы определяем позитивный lookahead, который говорит движку регулярного выражения искать The или the, за которыми следует fat.

"(T|t)he(?=\sfat)" => The fat cat sat on the mat.

*Отрицательный Lookahead*

Негативный lookahead используется, когда нам нужно получить все совпадения в строке, за которой не следует определенный шаблон. Негативный lookahead определяется так же, как и позитивный, с той лишь разницей, что вместо знака равенства мы используем знак отрицания !. Таким образом, наше выражение приобретает следующий вид: (?!...). Теперь рассмотрим (T|t)he(?!\sfat), что означает: получить все The или the в введенной строке, за которыми не следует слово fat, предшествующее знаку пробела.

"(T|t)he(?!\sfat)" => The fat cat sat on the mat.

*Положительный Lookbehind*

Положительный lookbehind используется для получения всех совпадений, которым предшествует определенный шаблон. Положительный lookbehind обозначается так: (?<=...). Например, регулярное выражение (?<=(T|t)he\s)(fat|mat) означает получить все fat или mat из строки ввода, которые идут после слова The или the.

"(? The fat cat sat on the mat.

*Отрицательный Lookbehind*

Отрицательный lookbehind используется для получения всех совпадений, которым не предшествует определенный шаблон. Отрицательный lookbehind обозначается выражением (?<!...). Например, регулярное выражение (?<!(T|t)he\s)(cat) означает: получить все cat слова из строки ввода, которые не идут после The или the.

"(? The cat sat on cat.

***Флаги***

Флаги также часто называют модификаторами, так как они могут изменять вывод regexp. Флаги, приведенные ниже являются неотъемлемой частью и могут быть использованы в любом порядке или сочетании regexp.

| Флаг | Описание |
| --- | --- |
| i | Нечувствительность к регистру: делает выражение нечувствительным к регистру. |
| g | Глобальный поиск: поиск шаблона во всей строке ввода. |
| m | Многострочность: анкер метасимвола работает в каждой строке. |

*Нечувствительные к регистру*

Модификатор i используется для поиска совпадений, нечувствительных к регистру. Например, выражение /The/gi означает прописную букву T, за которой следуют h и e. И в самом конце выражения стоит i, благодаря которому можно проигнорировать регистр. g применяется для того, чтобы найти шаблон во всей введенной строке.

"The" => The fat cat sat on the mat.

"/The/gi" => The fat cat sat on the mat.

*Глобальный поиск*

Модификатор используется для выполнения глобального поиска шаблона (поиск будет продолжен после первого совпадения). Например, регулярное выражение /.(at)/g означает любой символ, кроме новой строки, за которым следует строчный символ a, а затем t. Поскольку мы использовали флаг g в конце регулярного выражения, теперь он найдет все совпадения в вводимой строке, а не только в первой (что является стандартом).

"/.(at)/" => The fat cat sat on the mat.

"/.(at)/g" => The fat cat sat on the mat.

*Многострочный поиск*

Модификатор m нужен для выполнения многострочного поиска. Как было сказано раннее, привязки (^, $) используются для проверки, является ли шаблон началом или концом строки. Но если мы хотим, чтобы привязки работали в каждой строке, нужно использовать флаг m. Например, регулярное выражение /at(.)?$/gm означает: строчный символ a, за которым следует t и что угодно, только не новая строка. А благодаря флагу m этот механизм регулярных выражений соответствует шаблону в конце каждой строки строки.

"/.at(.)?$/" => The fat

cat sat

on the mat.

"/.at(.)?$/gm" => The fat

cat sat

on the mat.

***Жадный режим квантификатора***

Особенностью квантификаторов является возможность использования их в разных режимах: жадном, сверхжадном и ленивом. **Сверхжадный режим** включается добавлением символа «+» после квантификатора, а ленивый – символа «?».

"А.+а" // жадный режим

"А.++а" // сверхжадный режим

"А.+?а" // ленивый режим

По умолчанию квантификатор работает в жадном режиме. Это означает, что он ищет максимально длинное совпадение в строке. В результате выполнения этого кода:

**public** **static** **void** main(String[] args) {

String text = "Егор Алла Александр";

Pattern pattern = Pattern.compile("А.+а");

Matcher matcher = pattern.matcher(text);

**while** (matcher.find()) {

System.out.println(text.substring(matcher.start(), matcher.end()));

}

}

Вывод: Алла Алекс

Алгоритм поиска по заданному шаблону "А.+а", выполняется в следующей последовательности:

1. В заданном шаблоне первый символ – это русский символ буквы А. Matcher сопоставляет его с каждым символом текста, начиная с нулевой позиции. На нулевой позиции в нашем тексте находиться символ Е, поэтому Matcher перебирает последовательно символы в тексте, пока не встретит совпадение с шаблоном. В нашем примере это символ на позиции №5.



2. После того, как найдено совпадение с первым символом шаблона, Matcher сверяет соответствие со вторым символом шаблона. В нашем случае это символ «.», который обозначает любой символ.



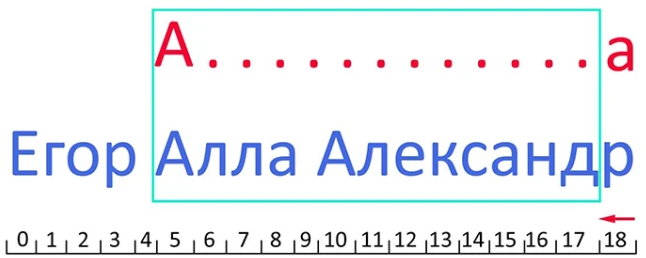
На шестой позиции – символ буквы л. Разумеется, он соответствует шаблону «любой символ».

3. Matcher переходит к проверке следующего символа из шаблона. В нашем шаблоне он задан с помощью квантификатора «.+». Поскольку количество повторений «любого символа» в шаблоне – один и более раз, Matcher берет по очереди следующий символ из строки и проверяет его на соответствие шаблону, до тех пор, пока будет выполняться условие «любой символ», в нашем примере – до конца строки (с поз. №7 -№18 текста).

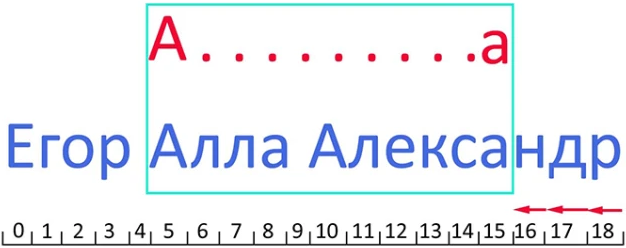


По сути, Matcher, захватывает все строку до конца – в этом как раз и проявляется его «жадность».

4. После того как Matcher дошел до конца текста и закончил проверку для части шаблона «А.+», Matcher начинает проверку для оставшейся части шаблона – символ буквы а. Так как текст в прямом направлении закончился, проверка происходит в обратном направлении, начиная с последнего символа:

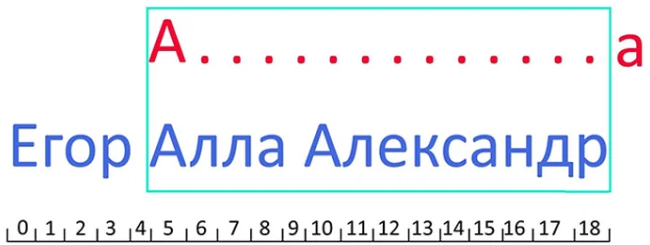


5. Matcher «помнит» количество повторений в шаблоне «.+» при котором он дошел до конца текста, поэтому он уменьшает количество повторений на единицу и проверяет соответствие шаблона тексту, до тех пор пока не будет найдено совпадение:



***Сверхжадный режим квантификатора***

В сверхжадном режиме работа матчера аналогична механизму жадного режима. Отличие состоит в том, что при захватывании текста до конца строки поиск в обратном направлении не происходит. То есть первые три этапа при сверхжадном режиме будут аналогичны жадному режиму. После захвата всей строки матчер добавляет остаток шаблона и сравнивает с захваченной строкой. В нашем примере при выполнении метода main с шаблоном "А.++а" совпадений не будет найдено.

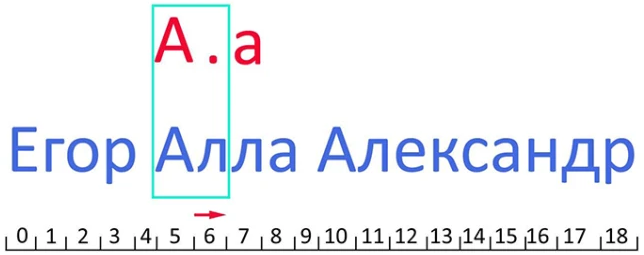


***Ленивый режим квантификатора***

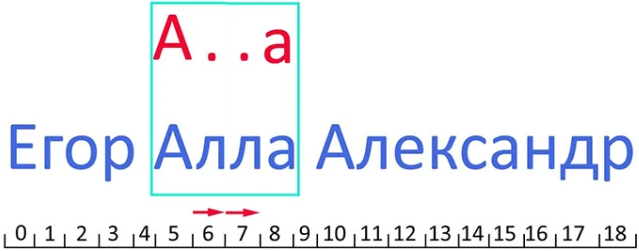
1. В этом режиме на начальном этапе, как и в жадном режиме, ищется совпадение с первым символом шаблона.

2. Далее ищется совпадение со следующим символом шаблона – любым символом.

3. В отличие от жадного режима, в ленивом ищется самое короткое совпадение в тексте, поэтому после нахождения совпадения со вторым символом шаблона, который задан точкой и соответствует символу на позиции №6 текста, Matcher будет проверять соответствие текста остатку шаблона – символу «а»



4. Поскольку совпадение с шаблоном в тексте не найдено (на позиции №7 в тексте находится символ «л»), Matcher добавляет еще один «любой символ» в шаблоне, так как он задан как один и более раз, и опять сравнивает шаблон с текстом на позициях с №5 по №8:



5. В нашем случае найдено совпадение, но конец текста ещё не достигнут. Поэтому с позиции №9 проверка начинается с поиска первого символа шаблона по аналогичному алгоритму и далее повторяется вплоть до окончания текста.



В результате работы метода main при использовании шаблона "А.+?а" мы получим следующий результат:

Алла

Алекса .

Как видно из нашего примера, при использовании разных режимов квантификатора для одного и того же шаблона мы получили разные результаты. Поэтому необходимо учитывать эту особенность и выбирать нужный режим в зависимости от желаемого результата при поиске.

***Экранирование символов в регулярных выражениях***

Поскольку регулярное выражение в Java, а точнее — его исходное представление задается с помощью строкового литерала, необходимо учитывать те правила спецификации Java, которые касаются строковых литералов. В частности, символ обратной косой черты «\» в строковых литералах в исходном коде Java интерпретируется как символ управляющей последовательности, который предупреждает компилятор, что следующий за ним символ — специальный и что его нужно особым образом интерпретировать.

String s=“The root directory is \nWindows”;

//перенос Windows на новую строку

String s=“The root directory is \u00A7Windows”;

//вставка символа параграфа перед Windows

Поэтому в строковых литералах, которые описывают регулярное выражение, и используют символ «\» (например, для метасимволов) **его нужно удваивать**, чтобы компилятор байт-кода Java не интерпретировал его по-своему.

String regex=”\\s”; // шаблон для поиска символов пробела

String regex=”\\”Windows\\””;

// шаблон для поиска строки ”Windows”

Двойной символ обратной косой черты также следует использовать для экранирования символов, задействованных в качестве специальных, если мы планируем их использовать как «обычные» символы.

String regex=”How\\?”; // шаблон для поиска строки “How?”

1. *Какие java-классы работают с регулярными выражениями? В каком пакете они расположены? Приведите пример анализа текста с помощью регулярного выражения и поясните код примера.*

Пакет java.util.regex исходно состоит из следующих трех классов:

- Pattern Class – объект класса Pattern представляет скомпилированное представление регулярного выражения. В классе Pattern публичный конструктор не предусмотрен. Для создания шаблона, вам сперва необходимо вызвать один из представленных публичных статичных методов compile(), который далее произведет возврат объекта класса Pattern. Регулярное выражение в данных методах принимается как первый аргумент.

- Matcher Class – объект класса Matcher представляет механизм, который интерпретирует шаблон, а также производит операции сопоставления с вводимой строкой. Аналогично классу Pattern, Matcher не содержит публичных конструкторов. Объект класса Matcher может быть получен путем вызова метода matcher() на объекте класса Pattern.

- PatternSyntaxException – объект класса PatternSyntaxException представляет непроверяемое исключение, которое обозначает синтаксическую ошибку в шаблоне регулярного выражения.

***Методы класса Pattern***

В классе Pattern есть и другие методы для работы с регулярными выражениями: **String pattern()** – возвращает исходное строковое представление регулярного выражения, из которого был создан объект Pattern:

Pattern pattern = Pattern.compile("abc");

System.out.println(Pattern.pattern())//"abc"

**static boolean matches(String regex, CharSequence input)** – позволяет проверить регулярное выражение, переданное в параметре regex на соответствие тексту, переданному в параметре input. Возвращает:

*true* – если текст соответствует шаблону;

*false* – в противном случае;

System.out.println(Pattern.matches("А.+а","Алла"));//true

System.out.println(Pattern.matches("А.+а","Егор Алла Александр"));//false

**int flags()** – возвращает значения параметра flags шаблона, которые были установлены при его создании, или 0, если этот параметр не был установлен.

Pattern pattern = Pattern.compile("abc");

System.out.println(pattern.flags());// 0

Pattern pattern = Pattern.compile("abc",Pattern.CASE\_INSENSITIVE);

System.out.println(pattern.flags());// 2

String[] split(CharSequence text, int limit) – разбивает текст, переданный в качестве параметра на массив элементов String. Параметр limit определяет предельное количество совпадений, которое ищется в тексте:

при limit>0 – выполняется поиск limit-1 совпадений;

при limit<0 – выполняется поиск всех совпадений в тексте

при limit=0 – выполняется поиск всех совпадений в тексте, при этом пустые строки в конце массива отбрасываются;

**public** **static** **void** main(String[] args) {

String text = "Егор Алла Анна";

Pattern pattern = Pattern.compile("\\s");

String[] strings = pattern.split(text,2);

**for** (String s : strings) {

System.out.println(s);

}

System.out.println("---------");

String[] strings1 = pattern.split(text);

**for** (String s : strings1) {

System.out.println(s);

}

}

Вывод на консоль:

*Егор Алла Анна*

*--------*

*Егор*

*Алла*

*Анна*

***Методы класса Matcher***

Matcher представляет собой класс, из которого создается объект для поиска совпадений по шаблону. Matcher – это «поисковик», «движок» регулярных выражений. Для поиска ему надо дать две вещи: шаблон поиска и «адрес», по которому искать. Для создания объекта Matcher предусмотрен следующий метод в классе Pattern: **рublic Matcher matcher(CharSequence input)** В качестве аргумента метод принимает последовательность символов, в котором будет производиться поиск. Это объекты классов, реализующих интерфейс CharSequence. в качестве аргумента можно передать не только String, но и StringBuffer, StringBuilder, Segment и CharBuffer. Шаблоном для поиска является объект класса Pattern, на котором вызывается метод matcher.

Pattern p = Pattern.compile("a\*b");

// скомпилировали регулярное выражение в представление

Matcher m = p.matcher("aaaaab");

//создали поисковик в тексте “aaaaab” по шаблону "a\*b"

Теперь с помощью нашего «поисковика» мы можем искать совпадения, узнавать позицию совпадения в тексте, заменять текст с помощью методов класса. Метод **boolean find()** ищет очередное совпадение в тексте с шаблоном. С помощью этого метода и оператора цикла можно производить анализ всего текста по событийной модели (осуществлять необходимые операции при наступлении события – нахождении совпадения в тексте). Например, с помощью методов этого класса **int start()** и **int end()** можно определять позиции совпадения в тексте, а с помощью методов **String replaceFirst(String replacement)** и **String replaceAll(String replacement)** заменять в тексте совпадения на другой текст replacement.

**public** **static** **void** main(String[] args) {

String text = "Егор Алла Анна";

Pattern pattern = Pattern.compile("А.+?а");

Matcher matcher = pattern.matcher(text);

**while** (matcher.find()) {

**int** start=matcher.start();

**int** end=matcher.end();

System.out.println("Найдено совпадение " + text.substring(start,end) + " с "+ start + " по " + (end-1) + " позицию");

}

System.out.println(matcher.replaceFirst("Ира"));

System.out.println(matcher.replaceAll("Ольга"));

System.out.println(text);

}

Вывод программы:

*Найдено совпадение Алла с 5 по 8 позицию*

*Найдено совпадение Анна с 10 по 13 позицию*

*Егор Ира Анна*

*Егор Ольга Ольга*

*Егор Алла Анна*

Из примера видно, что методы replaceFirst и replaceAll создают новый объект String – строку, представляющую собой исходный текст, в котором совпадения с шаблоном заменены на текст, переданный методу в качестве аргумента. Причём метод replaceFirst заменяет только первое совпадение, а replaceAll – все совпадения в тесте. Исходный текст остается без изменений.

Наиболее частые операции с регулярными выражениями при работе с текстом из классов Pattern и Matcher встроены в класс String. Это такие методы как split, matches, replaceFirst, replaceAll. Но на самом деле «под капотом» они используют классы Pattern и Matcher. Поэтому, если вам нужно заменить текст или сравнить строки в программе без написания лишнего кода, используйте методы класса String. Если же вам нужны расширенные возможности – вспомните о классах Pattern и Matcher.

1. *Что такое группы в регулярных выражениях? Как нумеруются группы? Что представляет собой группа номер 0(ноль)? Приведите пример с использованием групп регулярного выражения.*

***Группы сбора***

Группы сбора представляют способ обращения с несколькими символами как с одной единицей. Они создаются путем размещения символов, которые предстоит сгруппировать, в серии круглых скобок. К примеру, регулярное выражение (dog) составляет отдельную группу, содержащую буквы "d", "o", и "g".

Группы сбора нумеруются посредством определения числа открывающих круглых скобок слева направо. Так, в выражении ((A)(B(C))) присутствуют четыре подобные группы:

- ((A)(B(C)))

- (A)

- (B(C))

- (C)

Для определения числа групп, представленных в выражении, вызвать метод groupCount на объекте класса matcher в Java. Метод groupCount извлекает число типа int, отображающее количество групп сбора, представленных в сопоставляемом шаблоне.

Также имеется специальная группа, группа 0, которая во всех случаях представляет выражение в полном виде. Данная группа не включается в сумму, представленную методом groupCount.

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches { public static void main( String args[] ) {

// Строка для сканирования, чтобы найти шаблон

String str = "Крещение Руси произошло в 988 году! Не так ли?";

String pattern = "(.\*)([\\d+)(.\*)](\\\\d+)(.*))";

// Создание Pattern объекта

Pattern r = Pattern.compile(pattern);

// Создание matcher объекта

Matcher m = r.matcher(str); if (m.find( )) {

System.out.println("Найдено значение: " + m.group(0)); System.out.println("Найдено значение: " + m.group(1)); System.out.println("Найдено значение: " + m.group(2));

}else {

System.out.println("НЕ СОВПАДАЕТ"); } } }

|  |  |
| --- | --- |
| **Код выражения** | **Обозначение** |
| ^ | Соответствует началу строки. |
| $ | Соответствует концу строки. |
| . | Соответствует любому одиночному символу, за исключением новой строки. Использование опции m делает возможным соответствие новой строке. |
| [...] | Соответствует любому одиночному символу в квадратных скобках. |
| [^...] | Соответствует любому одиночному символу вне квадратных скобок. |
| \A | Начало целой строки. |
| \z | Конец целой строки. |
| \Z | Конец целой строки, за исключением допустимого терминатора конца строки. |
| re\* | Соответствует 0 либо более вхождений предыдущего выражения. |
| re+ | Соответствует 1 либо более вхождений предыдущего выражения. |
| re? | Соответствует 0 либо 1 вхождению предыдущего выражения. |
| re{ n} | Соответствует заданному n числу вхождений предыдущего выражения. |
| re{ n,} | Соответствует n или большему числу вхождений предыдущего выражения. |
| re{ n, m} | Соответствует n как минимум и m в большинстве вложений предыдущего выражения. |
| a| b | Соответствует a или b. |
| (re) | Группирует регулярные выражения и запоминает сравниваемый текст. |
| (?: re) | Группирует регулярные выражения, не запоминая сравниваемый текст. |
| (?> re) | Соответствует независимому шаблону без возврата. |
| \w | Соответствует словесным символам. |
| \W | Соответствует символам, не образующим слова. |
| \s | Соответствует пробелу. Эквивалент [\t\n\r\f]. |
| \S | Соответствует непробельному символу. |
| \d | Соответствует цифре. Эквивалент [0-9]. |
| \D | Соответствует нечисловому символу. |
| \A | Соответствует началу строки. |
| \Z | Соответствует окончанию строки. При наличии новой строки, располагается перед ней. |
| \z | Соответствует концу строки. |
| \G | Соответствует точке, где оканчивается предыдущее совпадение. |
| \n | Обратная ссылка на группу сбора под номером "n". |
| \b | Соответствует границе слова вне квадратных скобок. Соответствует возврату на одну позицию (0x08) внутри квадратных скобок. |
| \B | Соответствуют границам символов, не образующих слова. |
| \n, \t, etc. | Соответствует символам перевода строки, возврата каретки, табуляции, и т.д. |
| \Q | Управление (цитирование) всех символов до символа \E. |
| \E | Окончание цитаты, открытой при помощи \Q. |

**МЕТОДЫ КЛАССА MATCHER**

***Методы индексов*** представляют полезные значения индекса, которые демонстрируют точное количество соответствий, обнаруженных в вводимой строке.

|  |  |
| --- | --- |
| **№.** | **Метод и описание** |
| 1 | **public int start()** Возврат начального индекса к предыдущему совпадению. |
| 2 | **public int start(int group)** Возврат начального индекса к последовательности, захваченной данной группой в течение предыдущей операции установления соответствия. |
| 3 | **public int end()** Возврат позиции смещения следом за последним совпадающим символом. |
| 4 | **public int end(int group)** Возврат позиции смещения следом за последним символом к последовательности, захваченной данной группой в течение предыдущей операции установления соответствия. |

***Методы исследования***производят анализ вводимой строки и возврат булевого значения, отображающего наличие либо отсутствие шаблона.

|  |  |
| --- | --- |
| **№.** | **Метод и описание** |
| 1 | **public boolean lookingAt()** Предпринимает попытку поиска соответствия вводимой последовательности в начале области с шаблоном. |
| 2 | **public boolean find()** Предпринимает попытку поиска следующей подпоследовательности в вводимой последовательности, соответствующей шаблону. |
| 3 | **public boolean find(int start)** Сброс данного поиска соответствия и попытка поиска новой подпоследовательности в вводимой последовательности, соответствующей шаблону с указанного индекса. |
| 4 | **public boolean matches()** Предпринимает попытку поиска совпадений во всей области с шаблоном. |

***Методы замены*** представляют полезные методы для замены текста в вводимой строке.

|  |  |
| --- | --- |
| **№.** | **Метод и описание** |
| 1 | **public Matcher appendReplacement(StringBuffer sb, String replacement)** Производит нетерминальное присоединение и замену. |
| 2 | **public StringBuffer appendTail(StringBuffer sb)** Производит терминальное присоединение и замену. |
| 3 | **public String replaceAll(String replacement)** Заменяет каждую подпоследовательность в вводимой последовательности, совпадающей с шаблоном, указанным в замещающей строке. |
| 4 | **public String replaceFirst(String replacement)** Замещает первую подпоследовательность в вводимой последовательности, совпадающей с шаблоном, указанным в замещающей строке. |
| 5 | **public static String quoteReplacement(String s)** Возвращает литеральную замену Строки для указанной Строки. Данный метод производит сроку, которая будет функционировать в качестве литеральной замены s в методе appendReplacement класса Matcher. |

**Методы класса PatternSyntaxException**

***PatternSyntaxException*** представляет непроверяемое исключение, которое отображает синтаксическую ошибку в шаблоне регулярного выражения.

|  |  |
| --- | --- |
| **№.** | **Метод и описание** |
| 1 | **public String getDescription()** Представляет описание ошибки. |
| 2 | **public int getIndex()** Представляет индекс ошибки. |
| 3 | **public String getPattern()** Представляет шаблон регулярного выражения, содержащего ошибку. |
| 4 | **public String getMessage()** Производит возврат многострочной строки, содержащей описание синтаксической ошибки и ее индекс, ошибочный образец регулярного выражения, а также визуальную индикацию индекса ошибки в шаблоне. |